Application-Bypass Reduction for Large-Scale Clusters [*]

Adam Wagner, Darius Buntinas and Dhabaleswar K. Panda
Network-Based Computing Laboratory
Dept. of Computer and Information Science
The Ohio State University
{wagnera, buntinas, panda}@cis.ohio-state.edu

Ron Brightwell
Scalable Computing Systems Dept.
Sandia National Laboratories
bright@cs.sandia.gov

Abstract

Process skew is an important factor in the performance of parallel applications, especially in large-scale clusters. Reduction is a common collective operation which, by its nature, introduces implicit synchronization between the processes involved in the communication and is therefore highly susceptible to performance degradation due to process skew. A collective operation with application-bypass does not require the application to block in order for the operation to make progress. Application-bypass collective operations are therefore highly tolerant of skew. In this paper we describe the design and implementation of an application-bypass version of the reduction operation in MPICH over GM. We evaluate our implementation on a 16-node cluster. Under conditions of process skew we find a factor of improvement of up to 3.3 for our application-bypass reduction versus the default MPICH implementation. In addition, we see that this factor of improvement increases with system size, indicating that the application-bypass implementation is more scalable and skew-tolerant than the default non-application-bypass version. This framework promises design and development of high-performance and scalable collective communication libraries for next-generation large-scale clusters.

1. Introduction

When we visualize running a parallel application on a cluster, it's common to think of all processes involved in the computation executing in a synchronous manner. For example, it's natural to assume that all processes will start at the same instant. However, in reality processes may become unsynchronized or *skewed*. This may happen for a variety of reasons including heterogeneous systems consisting of nodes with different processing capabilities, varying communication latencies between nodes, unbalanced or asymmetric code where different nodes may be assigned tasks requiring different amounts of processing resources, and random effects such as interrupts or contention for resources between multiple processes on a given node. Process skew becomes more prevalent as the size of a cluster grows and more opportunities for unpredictable delays are introduced.

Process skew is an important factor in the performance of parallel applications, especially those involving collective communications. Collective communications [9][7] often by their nature introduce implicit synchronization in the form of communication dependencies between processes. Under conditions of process skew, these dependencies can cause some processes to wait idly for other processes to catch up. This results in ineffective CPU utilization, wasting resources that might otherwise be dedicated to useful processing.

Reduction is a common example of such a collective communication. In the default MPICH [8] implementation of reduction, each process involved in the communication calls the `MPI_Reduce` function. Internally, MPICH organizes the processes into a logical tree. Processes wait to receive messages from their children before sending a result to their parent and completing `MPI_Reduce`. So `MPI_Reduce` synchronizes the participating processes, requiring each process to wait until all processes below it in the logical tree have completed `MPI_Reduce`. This synchronization is not necessary for the majority of the processes involved in the communication. It would be more efficient if the reduction operation could make progress independently of the application, allowing parent processes to continue with other work until their child processes have sent their data. This technique is known as *application bypass* [2] and is discussed in detail in the next section.

This paper describes our design and implementation of an application-bypass version of the reduction operation in MPICH over GM [11]. We discuss the design challenges that we faced in the process of adapting the synchronous infrastructure provided by the default MPICH implementation to support our more flexible application-bypass operation. These challenges include the maintenance of intermediate reduction state, handling messages that arrive both earlier and later than normally expected and minimizing the overhead associated with the mechanisms that we chose to support asynchronous processing. We have evaluated our implementation and found a factor of improvement of up to 3.3 under conditions of process skew. Furthermore, we have observed that the factor of improvement increases with system size, indicating that our application-bypass implemen-

[*] This research is supported in part by a grant from Sandia National Laboratory (a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC004-94AL85000.), Department of Energy's Grant #DE=FC02-01ER25506, and National Science Foundation's grant #EIA-9986052.

tation is more scalable and skew-tolerant than the default non-application-bypass version.

The remainder of this paper is organized as follows. In the next section we discuss the basic concepts of application bypass and how they can be applied to the reduction operation. In Section 3 we provide an overview of GM and MPICH over GM. The design challenges we encountered while implementing our application-bypass reduction operation are discussed in Section 4 and then the details of our implementation are covered in Section 5. In Section 6 we evaluate the performance of our implementation and then we present our conclusions in Section 7.

2. Basic Concepts behind Application-Bypass Reduction

The goal in coding an application-bypass operation is to eliminate the need for applications to block while the operation makes progress. This sort of optimization is ideal for operations such as broadcast and reduction where there is no implied global synchronization between processes. It could even benefit synchronizing operations like barrier and all-reduce if they are implemented in a split-phase manner.

In MPICH, each process involved in a reduction calls the `MPI_Reduce` function at the application level to initiate the operation. Internally, `MPI_Reduce` organizes the processes into a logical binomial tree and the operation is then performed using point-to-point communication between processes. Figure 1 illustrates such a tree for eight processes. The root process is shown in black, internal processes are colored gray and leaf processes are shown in white. The arrows between processes indicate the direction of point-to-point messages associated with the reduction.

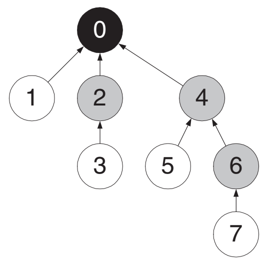


Figure 1. Example binomial tree used to organize point-to-point communications between eight processes involved in a reduction operation. The root node is shown in black, internal nodes are colored gray and leaf nodes are shown in white. The arrows between processes indicate the direction of messages involved in the reduction.

When calling `MPI_Reduce`, each process provides a buffer containing its input for the operation. The root process also provides an additional buffer to accept the operation results. While leaf processes simply need to send their input to their parents, all other processes must wait to receive results from their children before they can perform the arithmetic operation associated with the reduction. This organization introduces dependencies between processes. When processes become skewed, those which are parents in the tree may have to wait idly on children that are late. Application-bypass techniques eliminate the synchronous nature of these dependencies so that parent processes can proceed in spite of the late arrival of children at the `MPI_Reduce` point.

The default MPICH implementation could be enhanced using application-bypass techniques. The processes that can benefit from such enhancements are the internal ones. The behavior of the leaf processes need not be optimized as their only action is to perform a send to their parent. Similarly, the behavior of the root node can not benefit from optimization. Per the MPI standard, `MPI_Reduce` is implemented in a blocking fashion, so the root process expects the function call to return only when the reduction has completed across all processes. However, a split-phase implementation would enable optimization of the root node as well.

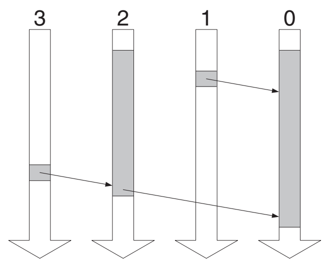
Figure 2 shows example time lines for a reduction involving four processes. Each large vertical arrow represents the progress of the operation for a given process. The portions of the large arrows shown in gray represent CPU utilization associated with the reduction. The small horizontal arrows represent point-to-point messages associated with the reduction. In this example, node zero is the root node, nodes one and three are leaf nodes and node two is an internal node. Note that the processes are slightly skewed, with nodes zero and two starting the reduction at approximately the same time, node one following shortly thereafter and node three being the last to begin.

Figure 2(a) shows the default non-application-bypass implementation. We can see that node two must wait idly on node three, which is late due to process skew. Figure 2(b) illustrates the application-bypass implementation. Here we can see that node two's reduction processing has been split into two components. The first portion is performed synchronously and is associated with the call to `MPI_Reduce`. Instead of waiting for the late child (node three), node two returns from `MPI_Reduce` and delegates the remainder of the reduction to asynchronous processing. The reduction operation resumes only when the message from node three finally arrives, and the time in between the synchronous and asynchronous portions can be utilized for other processing.

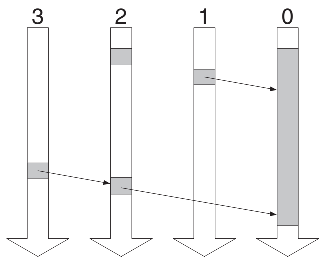
Under conditions of process skew, application-bypass techniques can reduce both the amount of time that processes spend waiting on each other and the amount of implicit synchronization associated with collective operations. These improvements can help reduce the amount of CPU utilization associated with the operation and increase the opportunity for overlap of communication and computation. The benefits of application-bypass operations are especially relevant in large-scale clusters where skew between processes becomes inevitable.

3. Overview of GM and MPICH over GM

GM [11] is a user-level message-passing subsystem for Myrinet networks. Myrinet [1] is a low-latency, high-bandwidth interconnection network that employs programmable network interface cards (NICs), cut-through



(a) Non-Application-Bypass



(b) Application-Bypass

Figure 2. Example time line for four processes involved in a reduction operation. The large vertical arrows represent the progress of the operation for each process. The gray portions of the large arrows represent CPU utilization associated with the reduction. Each small horizontal arrow represents a point-to-point message involved in the reduction.

crossbar switches and operating-system-bypass techniques to achieve full-duplex 2 Gbps data rates. GM consists of a lightweight kernel-space driver, a user-space library and a control program which executes on the NIC processor. The kernel-space code is only used for housekeeping purposes like allocating and registering memory. After taking care of such initialization tasks, the user-space library can communicate directly with the NIC-based control program, removing the operating system from the critical path.

MPI [10] is a standard interface for message passing in parallel programs. MPICH [8] is the reference implementation of MPI and has been ported to a variety of hardware platforms including GM over Myrinet. As previously mentioned, the standard MPICH implementation does not include application-bypass techniques. In order to illuminate the design challenges discussed in the next section, we will first highlight some of the relevant MPICH implementation details.

One such detail is the way in which MPICH handles the receipt of messages, both those which are expected by the application and those which are not. While there are multiple functions that may be used to receive messages with different semantics, we focus on the default case in this discussion. When a process is ready to receive a message, it calls the `MPI_Recv` function, providing criteria to identify the message to be received as well as an appropriate buffer for storage of the message. If a message arrives before a matching call to `MPI_Recv` has been made, MPICH allocates a temporary buffer, copies the message into the buffer and then adds it to the *unexpected queue*. When a process calls `MPI_Recv`, MPICH first searches the unexpected queue for a matching message. If a match is found, it simply copies the message from the unexpected queue to the buffer provided by the application. Otherwise, it polls the network until a matching message is received, at which point the message is copied into the application buffer and returned.

Another notable detail relates to the way GM uses memory when sending messages. GM can only send data located in memory which has been registered for DMA transfers (*pinned*). Since pinning and unpinning memory requires relatively expensive system calls, MPICH over GM uses two send modes to efficiently handle both small and large messages. Small messages are sent using *eager* mode and large messages are sent in *rendezvous* mode. Basically, in eager mode message data is copied into a pre-pinned buffer for sending, while in rendezvous mode the message data is pinned in-place and sent from its original location. Eager mode eliminates the overhead of pinning for small messages at the expense of a memory copy, while rendezvous mode eliminates the overhead of copying for large messages at the expense of pinning memory.

4. Design Challenges

This section discusses the design challenges we encountered while implementing application-bypass reduction. The specifics regarding our solutions to each issue will be addressed in detail in the next section.

4.1. Maintenance of Intermediate State

Recall that in order to minimize the impact of process skew, we would like to split the reduction processing into synchronous and asynchronous components. A requirement for this approach is the maintenance of intermediate state associated with the reduction. First, note that a parent node may have multiple children, each of which may be processed synchronously or asynchronously at different points in time. Therefore, we need to keep track of the running result of the reduction operation between the initial synchronous processing and potentially multiple periods of asynchronous processing.

Second, note that in addition to processing messages from children, internal nodes must also send their final result to their parent. However, this must not happen until all children have been processed. So we need a way to know when the processing of all children has completed and the send to the parent may be performed.

Also, if the last child processed is handled by the asynchronous portion of the code, then we need to be able to determine the appropriate parent associated with the reduction. The parent-child relationships between nodes can vary between reduction instances depending on which process is designated as the root of the reduction. A node's parent is calculated during the synchronous call to `MPI_Reduce` and must be recorded for potential use during asynchronous processing.

4.2. Handling Early Messages

Another challenge involved handling early or *unexpected* messages. The semantics for unexpected messages are simple in the default MPICH implementation. Because all reduction processing is performed synchronously, unexpected messages are simply those messages that arrive before the application calls `MPI_Reduce`. However, in our application-bypass implementation we need to perform some additional checking due to the asynchronous nature of the processing. First, as in the non-application-bypass case, the message must fail to match a receive associated with the synchronous processing in `MPI_Reduce`. Second, the message must also fail to satisfy a pending receive which is being managed asynchronously after exiting a call to `MPI_Reduce`. If the message matches a pending asynchronous receive, then it's actually a *late* message as opposed to an unexpected message, and must be handled appropriately as discussed below. Otherwise, the message is truly unexpected and must be saved for later processing.

4.3. Handling Late Messages

As mentioned above, *late* messages are those messages associated with a reduction operation that arrive after exiting a call to `MPI_Reduce`. These messages must be handled by the asynchronous component of our application-bypass implementation. So first, we need a way to differentiate these late messages from other messages and trigger the asynchronous processing. We also need to be able to match late messages to the proper reduction instance, as multiple reductions may be active concurrently and overlapped due to skew. For example, consider the eight-node case illustrated in Figure 1. Assume that our application performs several reductions back to back and that process six is consistently late in performing its send to process four. Each time process six is late, process four will delegate the associated operation to the asynchronous component of the implementation. Since there are several reductions performed back-to-back, there may be several outstanding receives from process six, each associated with a separate reduction instance. So when process four finally receives a message from process six, it needs to be able to match it to the appropriate reduction instance in order to maintain correctness.

4.4. Efficient Use of Interrupts

In order to support splitting the processing of reduction operations into synchronous and asynchronous components, some mechanism must be used to trigger the asynchronous processing upon receipt of late messages. One potential solution would involve using a dedicated thread to monitor incoming messages and activate the asynchronous processing as necessary. Another method would involve generating an interrupt upon the receipt of a late message. Both alternatives have benefits and disadvantages. The thread-based option would consume additional CPU resources while polling for late messages, but would not require the overhead of interrupts. The interrupt-based option would incur a certain amount of interrupt overhead with the arrival of late messages. However, this overhead would only occur when asynchronous processing is actually required, as opposed to the constant overhead of polling for late messages.

Based on our previous experience with the implementation of application-bypass broadcast [6], we decided to use an interrupt-based approach. Since interrupts incur a substantial performance penalty, this introduced another challenge in how to avoid the generation of unnecessary interrupts. For example, interrupts need not be generated while MPICH is already checking for receives within `MPI_Reduce`. They are also unnecessary if there are no outstanding children to be processed asynchronously. In this case, messages can be unexpected but not late. Also, note that interrupts are only required for internal nodes, as the root node must perform all of its processing synchronously and the leaf nodes have no children.

4.5. Reducing Frequency of Late Messages

As mentioned above, interrupts are not necessary if MPICH is already checking for receives while inside `MPI_Reduce`. We explored a potential optimization involving the addition of a small delay before exiting `MPI_Reduce` in the case where all children had not been processed. By delaying, we hoped to allow receives from the outstanding children to complete and thus avoid interrupts. The crucial decision here is how long to delay. If the delay is too short, then late children will not be able to catch up, but if the delay is too long, then unnecessary latency will be incurred.

We experimented with a simple scheme in which we calculated the delay based on the number of processes involved in the reduction. A more sophisticated scheme could be constructed by taking into account the position of the parent and child processes in the logical tree. However, such calculations become quite speculative when random skews are involved and we are still investigating these issues.

5. Our Implementation

In this section we present the details of our implementation of application-bypass reduction. The section is organized as follows. First we discuss the changes that we made to the MPICH infrastructure to support application-bypass processing. Next we walk through both the asynchronous and synchronous components of the processing to illustrate the associated logic.

5.1. Infrastructure Changes

First, we modified GM 1.5.2.1 to include the ability to generate signals from within the NIC-based control program. We added a new collective packet type for use when sending messages related to application-bypass reduction. In addition, we added the capability to disable and enable signals from within the MPICH layer via calls to the GM library. These two modifications are used together to minimize the number of signals that are generated. Signals are only generated by the NIC for messages of the new collective packet type, isolating them to only those situations where they are actually required. We initialize MPICH with signals in a disabled state, as initially there can not be any outstanding reductions. We only enable signals when outstanding reductions need to be processed asynchronously, and then again disable signals as soon as all outstanding reductions have been completed. Details on exactly how and when we choose to enable and disable signals are included below. When a signal is received by the host, it triggers the activation of the MPICH progress engine so that asynchronous processing may be performed.

The remainder of the changes were made to MPICH over GM version 1.2.4..8a. As mentioned previously, we needed to develop a strategy for handling both unexpected and late messages. We explored one solution which involved using the non-blocking versions of the MPICH send and receive primitives for internal point-to-point communication within the collective reduce operation. The default MPICH implementation uses the blocking versions of the send and receive primitives. By switching to the non-blocking versions we hoped to gain the extra control we needed to support asynchronous processing, while still re-using as much as possible of the existing MPICH infrastructure. While this solution did enable reuse of the existing MPICH message matching and queuing mechanisms, it also required the allocation and management of additional buffers for use in the non-blocking receives. In addition, it introduced extra complexity associated with trying to use the MPICH infrastructure in ways other than those in which it was intended to be used.

We instead chose to implement our own *unexpected queue* specifically for application-bypass messages. This enables us to manage unexpected messages in an efficient manner, reducing the maximum number of required message copies from two to one. It also prevents our optimizations from affecting the common case of non-collective point-to-point communications, which are left to the default MPICH mechanisms. In addition to the unexpected queue, we also added a *descriptor queue* to manage descriptors containing state information for pending reductions. Each descriptor includes the intermediate result of the reduction operation, the identity of the parent process to which results should be sent and a list of children from which receives are pending. The child list is also used for matching late messages to the appropriate entry in the descriptor queue (i.e. the appropriate reduction instance). Details on how both queues fit into our implementation are provided below.

5.2. Asynchronous Processing

Recall that in MPICH, each process involved in a reduction calls the `MPI_Reduce` function at the application level to initiate the operation. This call to `MPI_Reduce` is the synchronous component of our implementation. We also added code to enable pre-processing of incoming collective-communication packets before they are examined by the MPICH matching and queuing mechanisms. This pre-processing comprises the asynchronous portion of our implementation.

When a collective-communication packet arrives, we first check to see whether the current process is the root of the collective communication with which the packet is associated. If so, then no extra asynchronous action is taken. This is because the behavior of the root process is necessarily synchronous, so we can utilize the normal synchronous point-to-point communications. In such a case where we decide not to process a packet, it is handled by the default MPICH mechanisms.

If the current process is not the root, the descriptor queue is searched to see if the source of the packet matches an existing reduction. If so, the corresponding operation is performed and the descriptor is updated to reflect the fact that the child has been processed. If all children have been processed, the final result is sent to the parent and the descriptor is removed from the queue. If this action renders the queue empty (i.e. there are no outstanding reductions) then signals are disabled.

If the packet fails to match an entry in the descriptor queue, the message is added to the unexpected queue for later processing. Note that if the message is expected it is processed directly from the buffer associated with the packet, eliminating the need to copy it into a buffer associated with a point-to-point receive as in the default MPICH implementation.

5.3. Synchronous Processing

As mentioned above, the synchronous portion of our implementation takes place within `MPI_Reduce`. First, we determine whether or not to perform a given reduction in application-bypass mode. This decision is made based on the size of the message. If the size of the message is within the limits of eager-mode processing, we proceed in application-bypass mode. Otherwise, we simply perform a standard non-application-bypass reduction. We have not yet investigated a rendezvous-mode implementation due to the additional complexities involved in buffer management.

Assuming the reduction is being processed in application-bypass mode, we first ensure that signals are disabled as we will be explicitly making progress while inside `MPI_Reduce`. Next, we build a descriptor containing the intermediate state needed to manage the reduction operation or operations as well as a list of the child or children of the current process. This descriptor is added to a queue of outstanding reductions.

From this point onward, the reduction may actually be processed in parallel by both `MPI_Reduce` and the asynchronous routine. The logic within `MPI_Reduce` basically walks through the list of children in the reduce descriptor, checking for unexpected messages and making progress if pending receives are detected. When progress is made, the asynchronous portion of the code will process expected and late messages as detailed above. If an unexpected message from a child is encountered, the corresponding operation is performed and the associated descriptor is updated to reflect the fact that the child has been processed. If all children are processed within `MPI_Reduce`, the final result is sent to the parent and the descriptor is removed from the queue. If at the end of `MPI_Reduce` the descriptor queue is not empty, then signals are enabled.

Note that even though unexpected messages must be buffered in the unexpected queue, they are processed directly from the unexpected queue in `MPI_Reduce`, eliminating the need for a second copy to a buffer associated with a point-to-point receive as in the default MPICH implementation.

6. Experimental Results

We evaluated our implementation on a cluster of 16 quad-SMP 700-MHz Pentium-III nodes with 66-MHz/64-bit PCI. The nodes were connected via a Myrinet-2000 network consisting of PCI64B network interface cards with 2 MB of memory and 133-MHz LANai 9.1 processors connected to 16 ports of a 32-port switch. Our application-bypass implementation is based on MPICH 1.2.4..8a over GM 1.5.2.1 and all comparisons were performed against the original, unaltered software packages of the same versions.

We created a pair of microbenchmarks for use in evaluating our implementation. The first microbenchmark measures the overall latency of a reduction, while the second microbenchmark measures the CPU utilization associated with performing a reduction under conditions of varying process skew.

The latency benchmark works as follows. First, we determine the one-way message latency between the root node and the node which is furthest away from the root in the logical tree (the *last node*). Next, we time a series of 10,000 reductions and take the average, using a barrier to separate iterations. We start timing just before the last node begins the reduction. Then, when the root node completes the reduction, it sends a notification message to the last node, which stops timing and subtracts off the one-way latency associated with the notification message to determine the total reduction latency.

While overall latency is important, if we assume that conditions involving process skew will be common, a better measure of performance is the amount of CPU utilization associated with the reduction. Skew will inevitably increase the overall latency, but if we can reduce the CPU utilization, additional computation may be performed while the reduction completes asynchronously. For the CPU utilization benchmark, in addition to varying the number of nodes and the message size, we also introduce a variable amount of delay at each node to simulate process skew. First, we convert a given maximum amount of delay from microseconds to busy-loop iterations. All delays are then generated using busy loops as opposed to absolute timings so that the CPU utilization associated with asynchronous processing may be captured. Next, we perform a series of 10,000 reductions and take the average across all nodes, using a barrier to separate iterations.

Within each loop iteration, the timing measurements are taken as follows. We first start timing, then introduce a random amount of delay between zero and the maximum delay, perform the reduction, introduce a catchup delay and finally stop timing. The skew delay as well as the catchup delay are then subtracted from the measured time at each node to calculate the CPU utilization. The catchup delay is equal to the maximum skew delay plus a conservative estimate of the maximum reduction latency. The intent here is to be sure to delay long enough to capture all asynchronous processing in the overall time measurement.

The remainder of this section is organized as follows. First we present latency and CPU utilization results under conditions with no process skew, which is very optimistic in a large-scale cluster. Note that this is the worst-case scenario for our implementation, where we see all of the overhead involved in the application-bypass techniques. However, we next present results under conditions of varying process skew. These are the common conditions in large-scale clusters. Our solution is designed for such scenarios and we can clearly see its benefits over the default non-application-bypass implementation.

6.1. Results without Process Skew

Figure 3 shows the results of the latency benchmark for 2, 4, 8 and 16 nodes with zero process skew and single-element double-word messages. For small numbers of nodes, the latency of the application-bypass and non-application-bypass implementations are nearly identical. However, once the number of nodes increases past four, the asynchronous component of the application-bypass implementation begins to be utilized, resulting in the difference in latency due to overhead from signals. While not illustrated here, we have observed that this latency penalty remains fairly constant at increased message sizes.

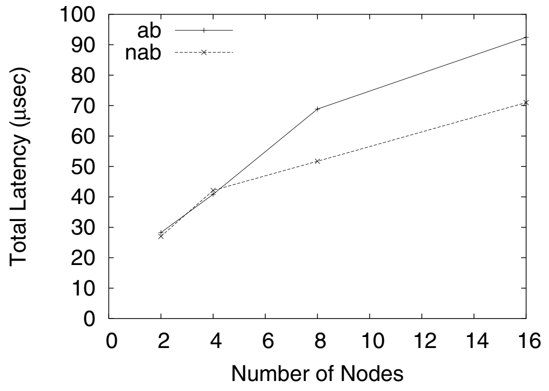
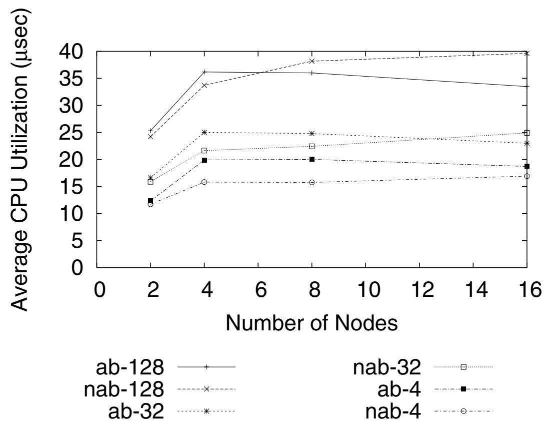
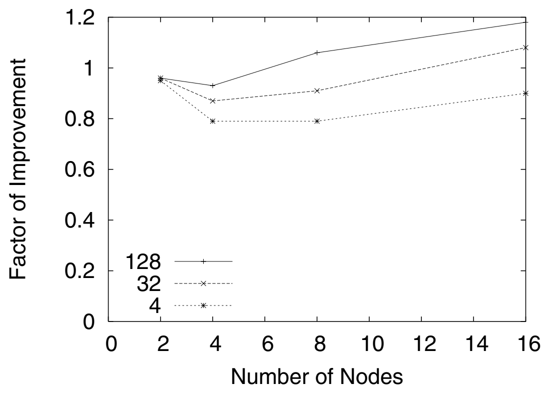


Figure 3. Average latency of application-bypass (ab) and non-application-bypass (nab) reduction for 2, 4, 8 and 16 nodes with zero process skew and single-element double-word messages.

Figure 4 shows the results of the CPU-utilization benchmark for 2, 4, 8 and 16 nodes with zero process skew and double-word messages of 4, 32 and 128 elements. We can see that as the number of nodes increases, the performance of the application-bypass implementation improves. Even for our relatively small 16-node cluster, the application-bypass implementation eventually outperforms the non-application-bypass implementation for all messages greater than four elements in size. This indicates that the application-bypass implementation is more scalable with respect to the number of nodes involved in an operation. We also see that the application-bypass implementation begins to outperform the non-application-bypass implementation at smaller numbers of nodes as the message size increases. This illustrates the benefit of the reduced number of message copies in the application-bypass implementation.



(a) Average CPU Utilization

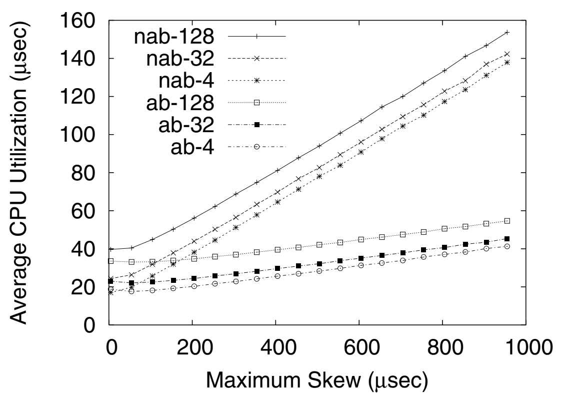


(b) Factor of Improvement

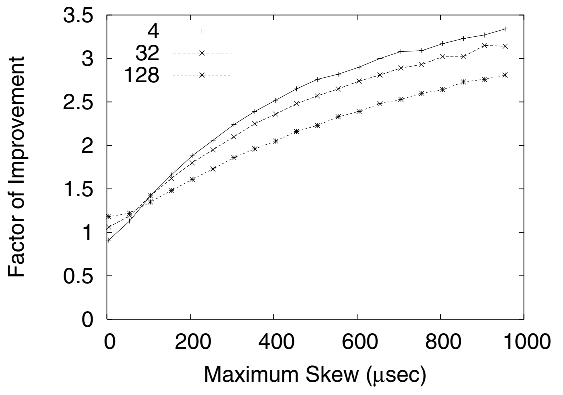
Figure 4. Average CPU utilization of application-bypass (ab) and non-application-bypass (nab) reduction for 2, 4, 8 and 16 nodes with zero process skew and 4, 32 and 128-element double-word messages.

6.2. Results with Process Skew

Figure 5(a) shows the results of the CPU-utilization benchmark for 16 nodes with increasing amounts of process skew and double-word messages of 4, 32 and 128 elements. We can see that the application-bypass implementation consistently outperforms the non-application-bypass implementation for all but the smallest amounts of skew. As the amount of skew increases, the non-application-bypass implementation spends more and more time polling the network for messages from late child nodes. However, the application-bypass implementation simply notes that there are pending receives from these late child nodes and then processes the messages asynchronously whenever they finally arrive. The overhead associated with signals in the application-bypass implementation is quickly overtaken by the overhead due to polling in the non-application-bypass implementation. Figure 5(b) shows a factor of improvement of 3.3 for four-element messages when the maximum skew is 1,000 μs.



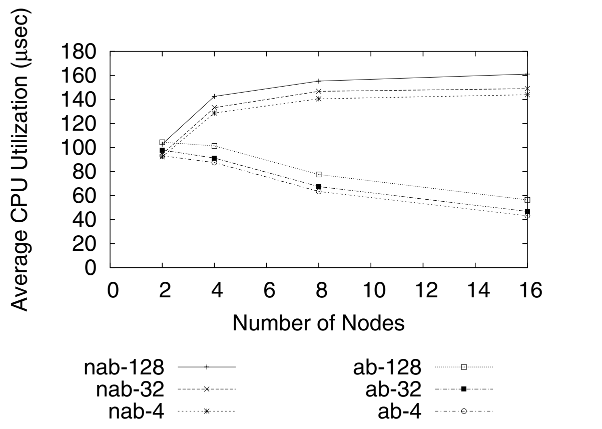
(a) Average CPU Utilization



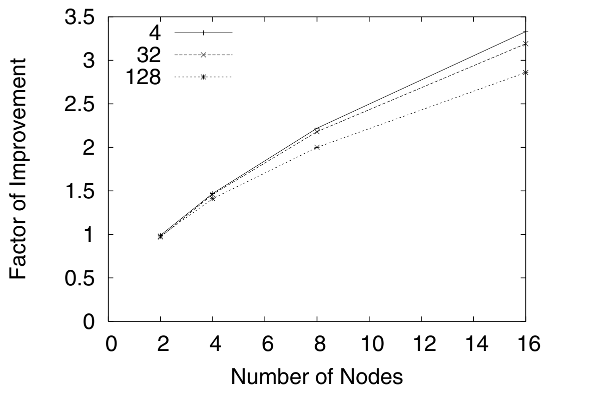
(b) Factor of Improvement

Figure 5. Average CPU utilization of application-bypass (ab) and non-application-bypass (nab) reduction for 16 nodes with varying process skew and 4, 32 and 128-element double-word messages.

Figure 6(a) shows the results of the CPU-utilization benchmark for 2, 4, 8 and 16 nodes with a maximum process skew of 1,000 μs and double-word messages of 4, 32 and 128 elements. These results confirm that the trends demonstrated in Figure 5 apply for varying numbers of nodes. Again, the application-bypass implementation consistently outperforms the non-application-bypass implementation with Figure 6(b) showing a maximum factor of improvement of 3.3 for 16 nodes and four-element messages. Furthermore, we can see that the factor of improvement increases with the number of nodes, demonstrating the enhanced scalability of the application-bypass implementation.



(a) Average CPU Utilization



(b) Factor of Improvement

Figure 6. Average CPU utilization of application-bypass (ab) and non-application-bypass (nab) reduction for 2, 4, 8 and 16 nodes with maximal process skew and 4, 32 and 128-element double-word messages.

7. Conclusions and Future Work

We have described both the design challenges and implementation details of our application-bypass version of reduction in MPICH over GM. Upon evaluation of our implementation, we found a factor of improvement of up to 3.3 when compared to the default non-application-bypass MPICH implementation under conditions of process skew. Furthermore, we note that the factor of improvement increases with system size, indicating that the skew-tolerant benefits of our application-bypass implementation will lead to better scalability than the non-application-bypass implementation on larger clusters.

In the future, we intend to evaluate the performance of application-bypass operations on large-scale clusters. We also intend to perform application-based evaluations to better understand how application-bypass solutions perform under real loads. Another area of investigation which we plan to pursue is the incorporation of NIC-based techniques [4][5][3] into our application-bypass implementations. Using NIC-based techniques, part or all of the operation may be performed on the NIC processor, as opposed to being performed on the host. This frees the host processor for use in other computation, naturally bypassing the application. Such abilities will deliver further advantages to the proposed framework.

Additional Information

Additional papers related to this research can be obtained from the Network-Based Computing Laboratory (http://nowlab.cis.ohio-state.edu) and Parallel Architecture and Communication Group (http://www.cis.ohio-state.edu/∼panda/pac.html) web pages.

References

[1] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. Seizovic, and W. Su. Myrinet - a gigabit per second local area network. In *IEEE Micro*, February 1995.

[2] R. Brightwell, R. Riesen, B. Lawry, and A. B. Maccabe. Portals 3.0: Protocol building blocks for low overhead communication. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters (CAC)*, April 2002.

[3] D. Buntinas and D. K. Panda. NIC-Based Reduction in Myrinet Clusters: Is It Beneficial? In *Proceedings of the SAN-02 Workshop (in conjunction with HPCA)*, February 2003.

[4] D. Buntinas, D. K. Panda, and P. Sadayappan. Fast NIC-based barrier over Myrinet/GM. In *Proceedings of the International Parallel and Distributed Processing Symposium 2001, (IPDPS)*, April 2001.

[5] D. Buntinas, D. K. Panda, and P. Sadayappan. Performance benefits of NIC-based barrier on Myrinet/GM. In *Proceedings of the Workshop on Communication Architecture for Clusters (CAC) held in conjunction with IPDPS '01*, April 2001.

[6] D. Buntinas and D. K. Panda and R. Brightwell. Application-Bypass Broadcast in MPICH over GM. In *Proceedings of the Cluster Computing and Grid Conference (CCGrid)*, May 2003.

[7] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach.* The IEEE Computer Society Press, 1997.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[9] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin/Cummings, 1994.

[10] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[11] Myricom. Myricom GM myrinet software and documentation. http://www.myri.com/scs/GM/doc/gm_toc.html, 2000.